**深 圳 大 学 实 验 报 告**

**课程名称： 计算机系统(2)**

**实验项目名称： Cache实验**

**学院： 计算机与软件学院**

**专业： 计算机科学与技术**

**指导教师： 马晨琳**

**报告人： 郑雨婷 学号： 2021150122 班级： 高性能**

**实验时间： 2023年5月11日至5月23日**

**实验报告提交时间： 2023年5月23日**

**教务处制**

|  |
| --- |
| **一、实验目的：**   1. 加强对Cache工作原理的理解； 2. 体验程序中访存模式变化是如何影响cahce效率进而影响程序性能的过程； 3. 学习在X86真实机器上通过调整程序访存模式来探测多级cache结构以及TLB的大小。 |
| **二、实验环境**  X86真实机器 |
| **三、实验内容和步骤**  **1、分析Cache访存模式对系统性能的影响**   * 1. 给出一个矩阵乘法的普通代码A，设法优化该代码，从而提高性能。   代码A：  #include <sys/time.h>  #include <unistd.h>  #include <stdlib.h>  #include <stdio.h>  int main(int argc, char \*argv[])  {  float \*a,\*b,\*c, temp;  long int i, j, k, size, m;  struct timeval time1,time2;    if(argc<2) {  printf("\n\tUsage:%s <Row of square matrix>\n",argv[0]);  exit(-1);  } //if  size = atoi(argv[1]);  m = size\*size;  a = (float\*)malloc(sizeof(float)\*m);  b = (float\*)malloc(sizeof(float)\*m);  c = (float\*)malloc(sizeof(float)\*m);  for(i=0;i<size;i++) {  for(j=0;j<size;j++) {  a[i\*size+j] = (float)(rand()%1000/100.0);  b[i\*size+j] = (float)(rand()%1000/100.0);  }  }    gettimeofday(&time1,NULL);  for(i=0;i<size;i++) {  for(j=0;j<size;j++) {  c[i\*size+j] = 0;  for (k=0;k<size;k++)  c[i\*size+j] += a[i\*size+k]\*b[k\*size+j];  }  }  gettimeofday(&time2,NULL);    time2.tv\_sec-=time1.tv\_sec;  time2.tv\_usec-=time1.tv\_usec;  if (time2.tv\_usec<0L) {  time2.tv\_usec+=1000000L;  time2.tv\_sec-=1;  }    printf("Executiontime=%ld.%06ld seconds\n",time2.tv\_sec,time2.tv\_usec);  return(0);  }//main  对代码进行优化，改动了**存储b矩阵时的结构以及读取b矩阵时的方式**：  #include <sys/time.h>  #include <unistd.h>  #include <stdlib.h>  #include <stdio.h>  int main(int argc, char \*argv[])  {  float \*a,\*b,\*c, temp;  long int i, j, k, size, m;  struct timeval time1,time2;    if(argc<2) {  printf("\n\tUsage:%s <Row of square matrix>\n",argv[0]);  exit(-1);  } //if  size = atoi(argv[1]);  m = size\*size;  a = (float\*)malloc(sizeof(float)\*m);  b = (float\*)malloc(sizeof(float)\*m);  c = (float\*)malloc(sizeof(float)\*m);  for(i=0;i<size;i++) {  for(j=0;j<size;j++) {  a[i\*size+j] = (float)(rand()%1000/100.0);  **b[j\*size+i] = (float)(rand()%1000/100.0);**  }  }    gettimeofday(&time1,NULL);  for(i=0;i<size;i++) {  for(j=0;j<size;j++) {  c[i\*size+j] = 0;  for (k=0;k<size;k++)  **c[i\*size+j] += a[i\*size+k]\*b[j\*size+k];**  }  }  gettimeofday(&time2,NULL);    time2.tv\_sec-=time1.tv\_sec;  time2.tv\_usec-=time1.tv\_usec;  if (time2.tv\_usec<0L) {  time2.tv\_usec+=1000000L;  time2.tv\_sec-=1;  }    printf("Executiontime=%ld.%06ld seconds\n",time2.tv\_sec,time2.tv\_usec);  return(0);  }//main   * 1. 改变矩阵大小，记录相关数据，并分析原因。   未优化：    优化后：    **2、编写代码来测量x86机器上（非虚拟机）的Cache 层次结构和容量**   1. 设计一个方案，用于测量x86机器上的Cache层次结构，并设计出相应的代码； 2. 运行你的代码获得相应的测试数据； 3. 根据测试数据来详细分析你所用的x86机器有**几级Cache**，**各自容量**是多大？ 4. 根据测试数据来详细分析**L1 Cache行**有多少？ |
| **四、实验结果及分析**  **1、分析Cache访存模式对系统性能的影响**  表1、普通矩阵乘法与及优化后矩阵乘法之间的性能对比   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 矩阵大小 | 100 | 500 | 1000 | 1500 | 2000 | 2500 | 3000 | | 一般算法执行时间 | 0.002992 | 0.405054 | 3.828211 | 16.839314 | 41.532330 | 94.712864 | 176.478773 | | 优化算法执行时间 | 0.002976 | 0.381535 | 3.148434 | 10.694446 | 25.921395 | 51.814044 | 88.550704 | | 加速比  speedup | 1.005376344 | 1.0616431 | 1.21590956 | 1.57458498 | 1.602241315 | 1.82793808 | 1.9929686 |   加速比定义：加速比=优化前系统耗时/优化后系统耗时；  所谓加速比，就是优化前的耗时与优化后耗时的比值。加速比越高，表明优化效果越明显。  分析原因：  优化前的算法核心代码如下：    可以看到在进行矩阵运算时，在获取b矩阵中的元素时，是按照列的顺序读取的，同列的元素存储位置并不相邻，有很差的空间局部性。所以不能够借助Cache完成数据传递，效率较低。  对代码进行优化，在存储b矩阵时，按列存储（将一行元素竖着存储）    这样在按行读取元素时，实际上读取的数据是一列元素，符合矩阵乘法运算的要求，且访问相邻元素可以有较好的空间局部性，效率提升。    将两组数据绘制成曲线放在一张图中对比：    可以看出，矩阵规模越大时，两种算法运行时间差异越大，同时根据表格，加速比也越高，表明优化效果越明显。  **2、测量分析出Cache 的层次结构、容量以及L1 Cache行有多少？**   1. 实验原理；   程序从不同Cache层级中读取数据的速率（即读吞吐量）是不同的，若速率有较大差异，说明在不同的Cache中，若速率相近，则在同一级Cache中。  测出L1 Cache的大小后，在此范围内确定Cache Line的大小，取不同的步长进行随机访问，步长大于Cache Line读吞吐量会下降。   1. 测量方案及代码；   测量Cache层次结构方案：  开辟一块大小为 size kb 的内存空间  进行若干次随机内存访问  记录时间，计算平均的数据吞吐量(kb/s)  画图分析，记录 size 与 kb/s 的关系  测量L1 Cache Line方案  开辟一块内存  按照不同的步长 stride 进行若干次内存访问  记录时间，计算平均的数据吞吐量(kb/s)  画图分析，记录 stride 与 kb/s 的关系  代码：  #include <bits/stdc++.h> #define B 1 #define KB 1024 #define MB 1048576 using namespace std; using std::chrono::high\_resolution\_clock; using std::chrono::duration; using std::chrono::duration\_cast; random\_device rd;//随机数生成 mt19937 gen(rd()); void random\_access(int size) {  int n = size / sizeof(char);  char\* buffer = new char[n];  fill(buffer, buffer+n, 1);  uniform\_int\_distribution<> dis(0, n-1);  int test\_times = 11451419 \* 10;  vector<int> random\_index;  for(int i=0; i<test\_times; i++){  int index = dis(gen);  random\_index.push\_back(index);  }  int sum = 0;  high\_resolution\_clock::time\_point t1 = high\_resolution\_clock::now();  for(int i=0; i<test\_times; i++){  sum += buffer[random\_index[i]];  }  high\_resolution\_clock::time\_point t2 = high\_resolution\_clock::now();  duration<double> time\_span = duration\_cast<duration<double>>(t2 - t1);  double dt = time\_span.count();  cout<<(((double)sum/1024.0) / dt)<<endl;  delete[] buffer; } void test1(){  vector<int> sizes**{**8\*KB,32\*KB,128\*KB,256\*KB,320\*KB,512\*KB,1024\*KB,2048\*KB,3072\*KB,4096\*KB,5120\*KB,6144\*KB,7168\*KB,8192\*KB,9216\*KB,10240\*KB,12000\*KB,13312\*KB,140000\*KB,15000\*KB**}**;  for(auto s : sizes){  random\_access(s);  } } void stride\_access(char\* buffer, int stride, int size){  int n = size / sizeof(char);  int sum = 0;  high\_resolution\_clock::time\_point t1 = high\_resolution\_clock::now();  for(int j=0; j<stride; j++){  for(int i=0; i<n; i+=stride){  sum += buffer[i];  }  }  high\_resolution\_clock::time\_point t2 = high\_resolution\_clock::now();  duration<double> time\_span = duration\_cast<duration<double>>(t2 - t1);  double dt = time\_span.count();  cout<<stride<<" "<<(((double)sum/1024.0) / dt)<<endl; } void test2(){  int size = 400 \* MB;  int n = size / sizeof(char);  char\* buffer = new char[n];  fill(buffer, buffer+n, 1);  vector<int> strides**{**1\*B,2\*B,4\*B,8\*B,16\*B,32\*B,64\*B,96\*B,128\*B,192\*B,256\*B,512\*B,1024\*B,1536\*B,2048\*B**}**;  for(auto s : strides){  stride\_access(buffer, s, size);  } } int main(){  //test1();  test2();  return 0; }   1. 测试结果；       表1 测量Cache层次结构数据    图2 测试Cache Line 大小      表2 测试Cache Line 大小数据    图1 测量Cache层次结构   1. 分析过程；   由图1可以看出，在8-320KB之间，吞吐量相近，对应L1 Cache，所以L1缓存的大小为320KB左右；在320KB-5120KB之间，吞吐量相近，对应L2 Cache，所以L2缓存的大小为5120KB左右；在5120KB-8192KB，吞吐量为一个层级，对应L3 Cache，所以L3缓存的大小为8192KB左右。这里有一点值得注意，每一级Cache都是上一级的一部分拷贝副本，也就是说，在5120KB时吞吐量下降，这个点就是L2 Cache的容量，而不是L1 Cache和L2 Cache的容量和。  由图2可以看出，在32-64B时吞吐量是明显下降的，说明步长大于Cache Line了，所以Cache Line的大小大概为32-64B。   1. 验证实验结果。     从进程管理器中，可以看出L1 Cache的大小为320KB,L2 Cache的大小为5MB，L3 Cache的大小为8MB，与分析结果一致，分析正确。    下载CPU-Z软件（我下载的是Ver .1.85.0，最新版本没找到在哪里查看一个line的大小）来查看L1 Cache Line的大小，看到为64B，与分析结果一致。 |
| **五、实验结论与心得体会**  本次实验的内容一中，我先运用cache的知识对矩阵乘法代码进行优化，通过优化前后的两个代码来测试，测试出程序运行时间、绘制成曲线进行对比，直观地看到了程序的空间局部性对效率的影响，当按照行顺序访问元素时，具有良好的空间局部性，因为它们地址上是相邻的，反之，按列顺序访问具有很差的空间局部性。  在第二个实验内容中，我让程序开辟不同大小的内存空间进行访问，通过吞吐量的不同来确定不同的Cache层次。更加直观地看到，距离CPU越近，速度越快，但容量越小。在这一步测量时，一开始的曲线和预期的相差很大，毫无规律可言，后来才注意到要将电脑上正在运行的其他程序（QQ、微信、浏览器等）都关掉，防止CPU抢占。 |

|  |
| --- |
| 指导教师批阅意见：  成绩评定：  指导教师签字：  2023年 月 日 |
| 备注： |